3

FUNDAMENTOS DE ANÁLISIS DE DATOS PYTHON

Un aspecto interesante antes de abordar tareas de predicción (clasificación o regresión) con lenguaje Python, es el hecho de conocer cómo se debe procesar la información (datos). Así mismo, es importante conocer el entorno de programación y su sintaxis. Debido a esto, este libro ha pretendido realizar una breve introducción a aquellos conceptos, recursos, librerías y sintaxis que son necesarios para preparar el conjunto de datos y seleccionar las variables más influyentes sobre un fenómeno o aspecto que se desee estudiar desde el área de la analítica de datos. Cabe mencionar que con el hecho de explicar unos conceptos básicos, los autores no pretenden llevar al lector al campo de la programación y elaboración de software con lenguaje Python, más bien llevarlo a una etapa inicial de preparación de datos para una futura implementación de algoritmos estimadores.

3.1 CONFIGURANDO EL ENTORNO PYTHON

Python es uno de los lenguajes de alto nivel de programación interpretado, de más rápido crecimiento para la aplicación de aprendizaje automático [44]. Es un lenguaje de programación versátil que se puede utilizar para muchos diferentes proyectos de programación. Por lo tanto, el primer paso es aprender cómo instalarlo. Para ello, hay varias opciones. Una de ellas es hacerlo desde la página **https://www.python.org/**. Otra opción recomendada por los autores es hacerlo desde la página web de Anaconda **https://www.anaconda.com/** y hacer clic en el botón "Free Downloads" (Figura 14). Representa una distribución de código abierto que reúne los lenguajes Python y R, y se ha consolidado como una herramienta fundamental en el campo de la ciencia de datos y el aprendizaje automático (*Machine Learning*). Esta plataforma es esencial para una variedad de aplicaciones, que van desde el

procesamiento de grandes conjuntos de datos hasta el análisis predictivo y los cálculos científicos. Su enfoque central radica en simplificar la instalación y gestión de paquetes de software, lo que resulta en una mayor eficiencia y facilidad de uso para profesionales y entusiastas de estos campos.

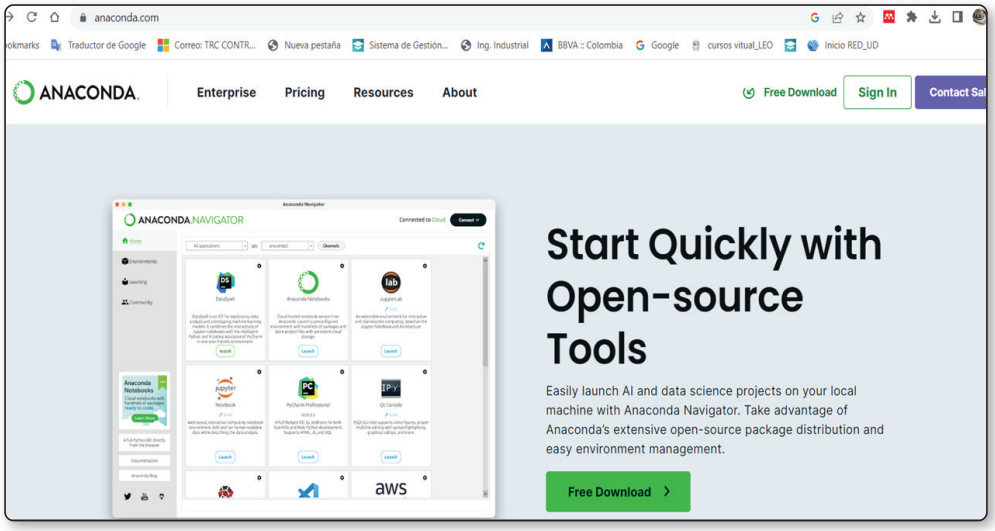


Figura 14. Web de Anaconda. Fuente: Anaconda

A continuación, hacer clic en el instalados descargado. El proceso de descarga comenzará automáticamente, y una vez que la descarga se haya completado, se procede a instalar Python en la computadora (Figura 15). Es importante tener en cuenta que este proceso puede llevar algo de tiempo, por lo que se debe tener paciencia durante la instalación.

Figura 15. Anaconda para Windows. Fuente: Anaconda

Una vez completada la instalación en el equipo, se abrirá una ventana similar a la que se muestra, que incluye varios Entornos de Desarrollo Integrados (IDE, por sus siglas en inglés, Integrated Development Environment). Es importante recordar que Python es un lenguaje de programación y que para trabajar con él se utilizará el IDE Jupyter Notebook, reconocido por ser un entorno amigable que no solo permite escribir código [45]. Para comenzar a trabajar, simplemente se hace clic en "Launch", o alternativamente se puede activar buscando "Jupyter" desde el menú de inicio de Windows (Figura 16).

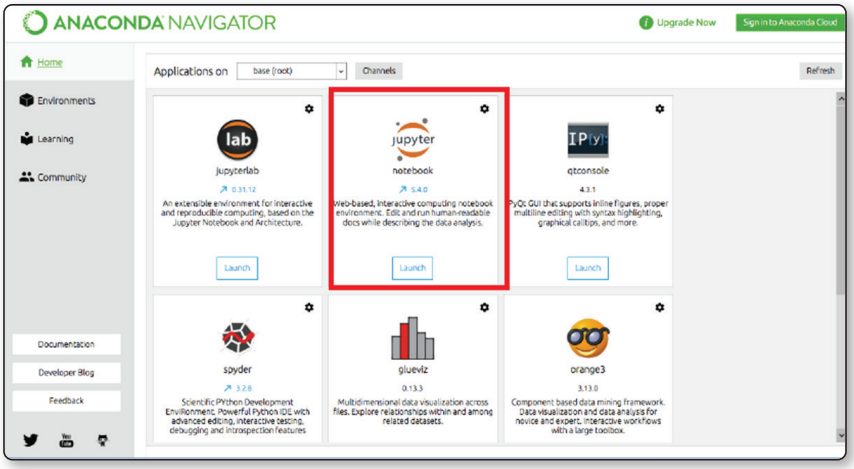


Figura 16. Apertura de entorno Jupyter

Al abrir el entorno Jupyter Notebook, se despliega una ventana similar a la que se muestra en la figura, que presenta las diversas carpetas y archivos almacenados en el PC, simulando un explorador de archivos de Windows. Por consiguiente, para iniciar un nuevo proyecto, se requiere seleccionar la ubicación donde se desea guardar este archivo. En este caso de ejemplo, el proyecto se guardará en la carpeta "*Machine Learning*"(Figura 17).

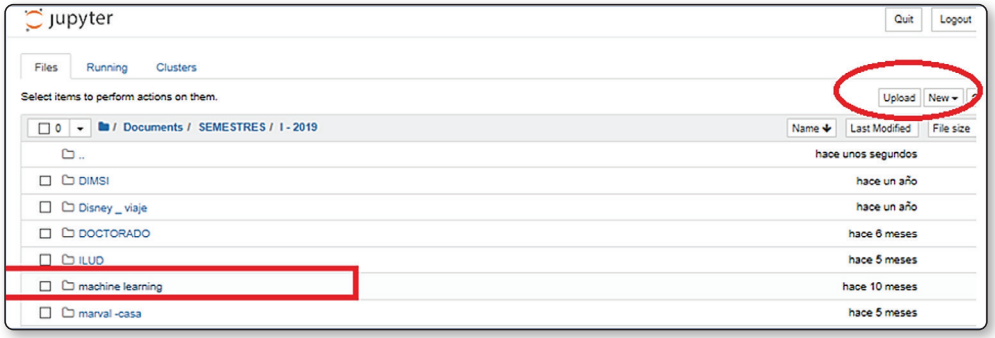


Figura 17. Lugar donde se guardará el primer archivo en Jupyter

Ahora que se ha decidido dónde guardar el archivo con el que se trabajará, el siguiente paso es crear el primer archivo de trabajo, el cual será el espacio donde se programará en Python. Para hacer esto, se debe seleccionar el botón "New" ubicado en la esquina superior derecha de la ventana mostrada. Al hacer clic en este botón, se desplegará una nueva ventana con la opción "Python 3", lo que indica que se ha creado el archivo de trabajo. Es importante tener en cuenta que este primer archivo creado no tiene un nombre definido por el usuario y aparecerá con el nombre por defecto "Untitled" (Figura 18).

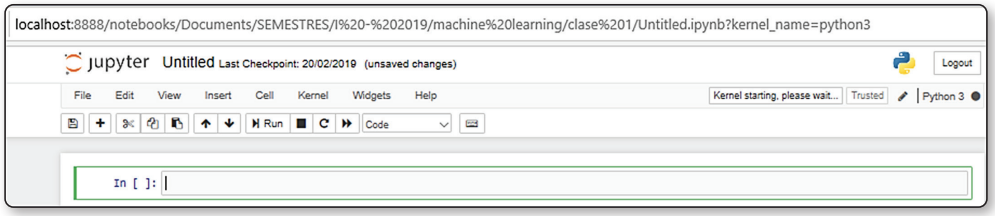


Figura 18. Primer archivo en Jupyter sin un nombre definido

Para cambiar el nombre del archivo recién creado, debe dirigirse a la barra de menús y seleccionar la opción "File". Al hacerlo, se desplegará una ventana emergente donde es posible seleccionar la función "Rename" (Figura 19). Esta acción simplifica la tarea de personalizar el nombre del archivo según nuestras necesidades y preferencias.

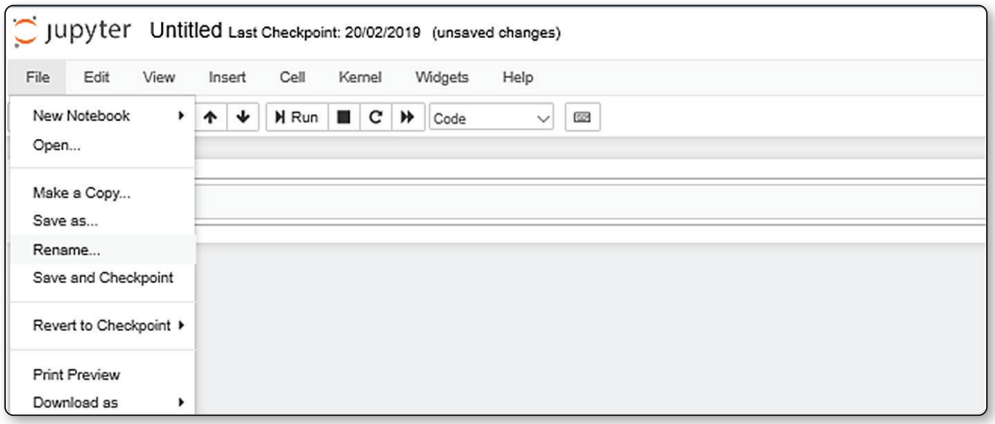


Figura 19. Cambio de nombre al archivo en Jupyter

De tal manera que se muestra una ventana para escribir el nombre del archivo. Para este ejemplo, se ha llamado "mi primer proyecto". De esta manera ya está listo el entorno para empezar a trabajar en lenguaje Python dentro de un entorno de trabajo Jupyter (Figura 20).

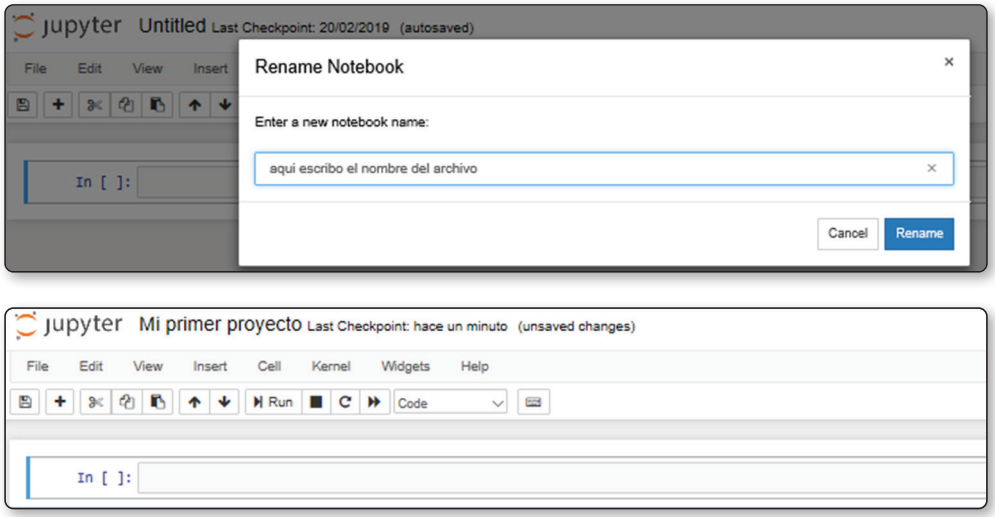


Figura 20. Cambio y visualización del nuevo nombre del primer archivo en Jupyter - Python

Si se desea saber si efectivamente el archivo de trabajo *"Mi primer proyecto"* quedó bien creado, se puede abrir el explorador de archivos de Windows, buscar la carpeta donde se guardó el proyecto o análisis de datos, que en este caso fue la carpeta *"Machine Learning"* y dentro de esta debe estar creado el archivo *" Mi primer proyecto*" como se muestra (Figura 21).

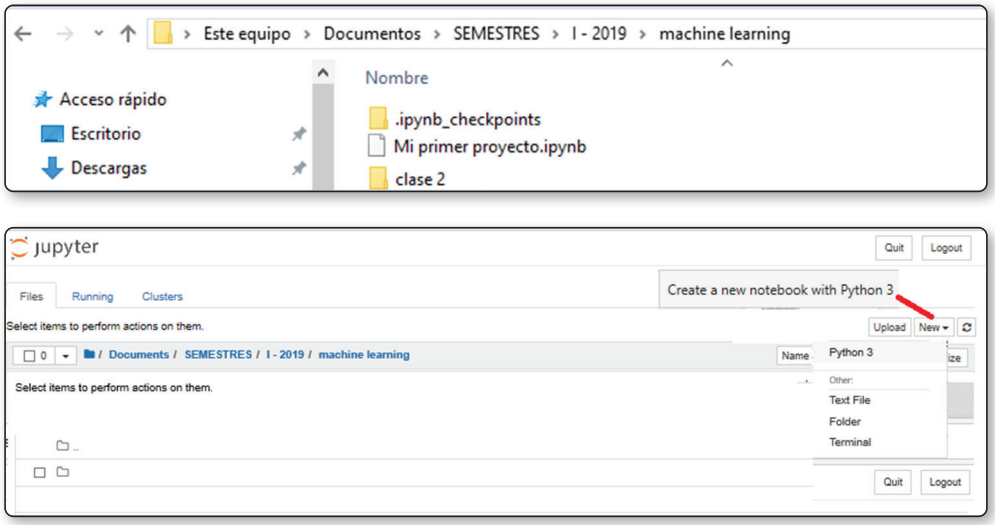


Figura 21. Verificación de la carpeta y archivo creado en Jupyter - Python

3.2 LIBRERÍAS BÁSICAS DE PYTHON

Antes de proceder con las librerías básicas, hay que mencionar ciertas características del lenguaje de programación Python [46]:

- Distingue entre mayúsculas y minúsculas.
- Se puede usar hash (#) al inicio de cada línea de código para hacer comentarios.
- Se usan espacios en blanco para indicar bloques de código (los espacios en blanco son importantes).

Python posee diversas librerías para diferentes usos, desde análisis de datos estadísticos hasta predicciones por medio de herramientas de análisis de datos. Las más usuales se describen en la Figura 22.

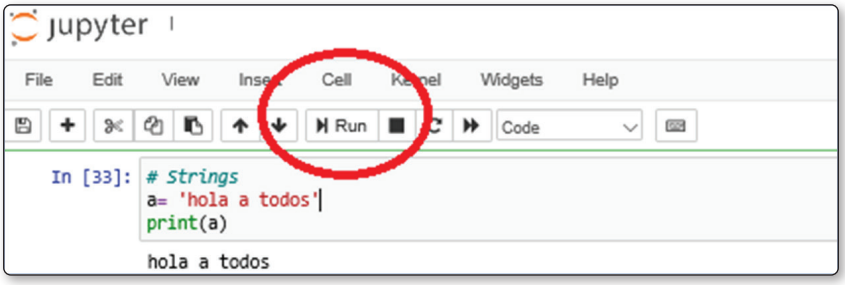
Figura 22. Descripción de algunas librerías de Python

En el presente libro se hace uso de las mencionadas librerías, aunque cabe mencionar que existen otras que pueden realizar acciones similares a las implementadas con código Python con estas librerías.

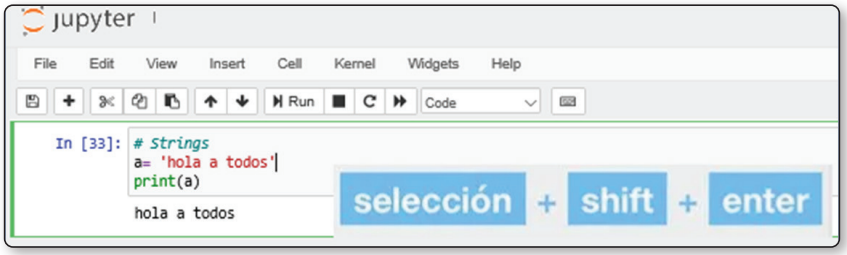
3.3 BÁSICO DE SINTAXIS DEL LENGUAJE PYTHON

Antes de trabajar tareas de clasificación (predicción), regresión o agrupamiento por ejemplo con algoritmos de *Machine Learning* en lenguaje Python, es conveniente saber un poco sobre la sintaxis usada por este lenguaje. A continuación, se muestra algo básico al respecto. Son ejemplos sencillos con los que se pretende que el lector conozca algunas instrucciones o secuencia de pasos acerca de la programación con Python que más adelante le puede ser útil para comprender como el código que se mostrará ejecuta diversas acciones para obtener un resultado. Se muestran ejemplos de sintaxis y de las tres librerías más usadas para la preparación y selección de características con Python (pandas, Numpy y Matplotlib). Estas mismas suelen usarse en algunos algoritmos de *Machine Learning* y pueden ser integradas con otras librerías más especializadas como scikitlearn, tensor Flow, keras, Seaborn, entre otras [47].

Para iniciar se escribirá una línea de código en Python la cual se mostrará cómo debe ejecutarse para visualizar. Existen dos formas. La primera es haciendo clic en el botón *"Run"* mostrado.



Otra opción es dirigirse a la celda que contiene el código que se desea ejecutar y presionar simultáneamente las teclas "Shift" + "Enter".



3.3.1 Strings y numéricos

Las cadenas en Python o strings son un tipo inmutable que permite almacenar secuencias de caracteres. Para esto se asigna un texto a la variable, por ejemplo en este caso la llamada variable "a". Se usa *"print"* para mostrar el resultado. Obsérvese que la cadena debe usarse con comillas sencillas (') o con comillas dobles (") una al inicio y otra la final de la cadena. También es posible asignar un valor numérico a una variable como por ejemplo la variable "X" en cuyo caso el valor numérico no debe estar en comillas.

```
# Asignación de string
a= 'Esto es una cadena o string'
print(a)

print('-------------')

# Asignación Numérica
X= 1000
print(X)
```

```
'Esto es una cadena o string'
-------------
1000
```

Una situación que muchas veces se puede dar, es cuando se quieren asignar múltiples valores numéricos o string a varias variables usando una sola línea de código. En este caso se procede como se muestra a continuación donde se les asigna a las variables x, y, z sus valores.

```
# Asignación múltiple de string variables
(x, y, z) = 10, 20, 30
print('Los valores asignados a las variables x, y, z son:' , x, y, z )
print('-------------')

# Asignación múltiple numérica
x, y, z = "YO ", "soy", "colombiano"
print('Las variables asignadas x, y, z tomaron valores:' , x, y, z )
```

```
Los valores asignados a las variables x, y, z son: 10 20 30
-----------------------------------------------------
Las variables asignadas x, y, z tomaron valores: YO soy colombiano
```

A menudo, nos encontramos con situaciones en las que la cadena que queremos imprimir es muy extensa, lo que puede hacer más legible dividirla en dos líneas. Esto se puede lograr asignando la cadena a una variable y utilizando otros elementos adicionales. Asimismo, podemos incluir un salto de línea dentro de la cadena misma, lo que indica que el texto posterior al salto será impreso en una nueva línea.

```
s = "Primer línea y Segunda linea juntos"
print (s)

print ("------")
p = "Primer línea \n Segunda linea"
print (p)
```

```
Primer línea y Segunda línea juntos
------
Primer línea
```

Segunda línea En numerosas ocasiones al desarrollar un programa, es necesario imprimir una cadena que incluya valores numéricos como resultado de algún proceso ejecutado. En este contexto, es crucial considerar la sintaxis adecuada para lograrlo. Tal situación puede presentarse cuando, por ejemplo, se desea que el resultado imprima una cadena con datos numéricos provenientes de una variable en su interior. A continuación, se muestra un ejemplo de cómo concatenar una cadena y un valor numérico a la variable "X":

```
# Asignación Numérica
X= 1000
print(X)
print (str(X) + 'pesos colombianos')
```

```
1000 pesos colombianos
```

Pero también existen otras maneras de trabajar uniendo string y números conocidos con el termino de "formatear una cadena" la cual se realiza mediante el símbolo %. En la parte anterior (izquierda) de este símbolo se coloca un símbolo cómo %s (imprimir cadena), %d (imprimir número decimal entero), %d (imprimir número decimal de coma flotante), que indica el tipo variable que se quiere imprimir, y por otro a la derecha del % tenemos la variable a imprimir.

```
# Formateo de cadenas con una variable numérica con %
X = 8
p = "El resultado es el número: %d" % X
print (p)
print  ('---------')

# Formateo de cadenas con dos variables numéricas con %
p = "El resultado es la suma de estos dos números %d y %d." % (8, 18)
print (p)
print  ('---------')

# Formateo de cadenas con variables numéricas y una variable string con %
nombre = 'Leonardo'
numero = 42
p= '%s %d' % (nombre, numero)
print (p)
```

```
1000
1000 pesos colombianos
---------
El resultado es el número: 8
---------
El resultado es la suma de estos dos números 8 y 18.
---------
Leonardo 42
```

Una alternativa para formatear cadenas en Python es emplear el método format(), el cual inserta los valores dentro del string o marcador de posición de la cadena definido mediante llaves: {} como se ilustra en los siguientes ejemplos.

```
# Formateo de cadenas con dos variables numéricas con format().
k = "Los números son {} y {}".format(5, 10)
print (k)
print  ('---------')

ejemplo1 = "Mi nombre es {nombre} y tengo {año} años".format (nombre =
"John", año = 36)
ejemplo2 = "Mi nombre es  {0}    y tengo {1}  años".format("John",36)
ejemplo3 = "Mi nombre es  {}    y tengo  {}  años".format("John",36)
print (ejemplo1)
print (ejemplo2)
print (ejemplo3)
```

```
Los números son 5 y 10
---------
Mi nombre es John y tengo 36 años
Mi nombre es  John    y tengo 36  años
Mi nombre es  John    y tengo  36  años
```

3.3.2 Booleanos

Tal como ocurre en diversos lenguajes de programación, en Python también se encuentra el tipo de dato bool o booleano, que posibilita el almacenamiento de dos valores fundamentales: True o False. Estos valores booleanos son esenciales en la programación ya que permiten la evaluación de condiciones y la toma de decisiones en base a la veracidad o falsedad de una afirmación [48]. Por lo general son usados con los condicionales if, else que se mencionaran más adelante. Por ahora es importante mencionar que Python posee una instrucción que permite identificar el tipo dato asignado a una variable. Esto se realiza mediante type () como se muestra en el siguiente fragmento de código para varias variables.

```
#Asignación de booleanos a las variables a y b
a = True
b = False
print(a, b)
print('-------------')
#Tipo de variable
a = 13
type(a)
print('-------------')

#Tipo de variable
b = 13.5
type(b)
print('-------------')

#Tipo de variable
c = "hola"
type(c)
print('-------------')

#Tipo de variable
c = "hola"
type(c)
print('-------------')
```

```
#Tipo de variable
d = False
type(d)
print('-------------')
```

```
True False
------------------------
Int
------------------------
Float
------------------------
Str
------------------------
Bool
```

Un valor booleano puede, además, ser el producto de la evaluación de una expresión. Algunos operadores, como el de mayor que, menor que o igual que, generan un valor bool al comparar dos elementos. Esta característica es esencial en la programación, ya que permite realizar comparaciones y tomar decisiones basadas en la relación entre valores numéricos u otros tipos de datos. Por ejemplo, al comparar dos números, si el primer número es mayor que el segundo, la expresión generará el valor True; de lo contrario, devolverá False.

```
# Evaluar expresiones mediante un booleano
print (1 > 0)  #True
print (1 <= 0) #False
print (9 == 9) #True
```

```
True
False
True
```

3.3.3 Aritmética

A continuación, se muestra cómo se realizan diferentes operaciones aritméticas con lenguaje Python. El paso para el lector consiste en ejecutar el código para visualizar los resultados.

```
#- Operaciones aritméticas

z = 5 + 2   #Suma
print(z)
```

```
z = 5 - 2  #Resta
print(z)

z = 5 * 2  #Multiplicación
print(z)

z = 5 / 2  #División
print(z)

z = 5 ** 2 #Potenciación
print(z)

z = 5 % 2  #Residuo de una división
print(z)

z = 5 // 2  #Parte entera de una división
print(z)
```

3.3.4 Listas

La lista es un tipo de secuencia de datos (flotantes, enteros o string) que usan la notación de corchetes. Suele usarse para implementar el código Python de los algoritmos de *Machine Learning*. Las listas en Python son uno de los tipos o estructuras de datos más versátiles del lenguaje, ya que permiten almacenar un conjunto arbitrario de datos [47]. En los siguientes fragmentos de código se pueden apreciar algunos de sus diversos usos.

```
#Lista formada por diversos tipos de datos
Lista =[1, 2, 3, “perro”, “gato”]
print(Lista)
print(‘-------------’)

# Indexación y cantidad de elementos de la lista
Lista =[1, 2, 3, “perro”,”gato”,”ratón”]
print(Lista [0])
print(Lista [4])

# Cantidad de elementos de la lista
len(Lista)
print (“La cantidad de elementos de la lista son:”, len (Lista))
print(‘-------------’)
```

```
[1, 2, 3, 'perro', 'gato']
------------------------
1
gato
```

La cantidad de elementos de la lista son: 6

En una lista así como es posible agregarle elementos, también Python permite su eliminación. Si tenemos una lista a con 6 elementos almacenados en ella, podemos acceder a los mismos usando corchetes y un índice, que va desde 0 a n-1 siendo n el tamaño de la lista. En el siguiente ejemplo se muestra cómo podría realizarse. Se continúa con la lista creada anteriormente.

```
Lista =[1, 2, 3, "perro","gato","ratón"]
del(Lista [0])                    # eliminar el elemento "o" la lista
del(Lista [-1])                   # eliminar el último elemento de la lista
Lista.remove('gato')        # elimina el elemento llamado "gato" de la
lista

print(Lista)
print ("La cantidad de elementos de la lista son:", len (Lista))
```

```
[2, 3, 'perro']
La cantidad de elementos de la lista son: 3
```

De igual manera que otros elementos claves de la programación con Python, sobre las listas se pueden ejecutar diversas acciones, como crear sublistas más pequeñas de una más grande. Para ello debemos de usar: entre corchetes, indicando a la izquierda el valor de inicio, y a la izquierda el valor final que no está incluido. Algunas de las más populares se muestran en los siguientes fragmentos de código.

```
# Obtener una lista a partir de otra lista
Lista =[1, 2, 3, "perro","gato","raton"]
fraccion_lista = Lista [0:3]
print(fraccion_lista)
print ("La cantidad de elementos de la lista son:", len (Lista))
print ("La cantidad de fracción de lista son:", len (fraccion_lista))
print('-------------')
```

```
[1, 2, 3]
La cantidad de elementos de la lista son: 6
La cantidad de fracción de lista son: 3
```

Algunos de los métodos usuales para trabajar con listas son los que se muestran en los siguientes fragmentos de código.

```
# Método append() añade un elemento al final de la lista.
Lista =[1, 2, 3, “perro”,”gato”,”ratón”]
Lista.append(40)
print (“Método append()”)
print (Lista)
print()

#Método extend() permite añadir una lista a la lista inicial.
Lista =[1, 2, 3, “perro”,”gato”,”ratón”]
Lista1 = [“a”,”b”,”c”]
Lista.extend(Lista1)
print (“Método extend() “)
print(Lista)
print()

#Método insert() añade un elemento en una posición o índice determinado.
Lista =[1, 2, 3, “perro”,”gato”,”ratón”]
Lista.insert(1, “X”) # En la posición 1 de la Lista se meterá el valor X
print (“Método insert()”)
print(Lista)
print()

#Método remove() recibe como argumento un objeto y lo borra de la lista.
Lista =[1, 2, 3, “perro”,”gato”,”ratón”]
Lista.remove(3)
print (“Método remove()”)
print(Lista)
print()

#Método sort() ordena de forma ascendente la lista numerica o de string.
ListaS =[“perro”,”gato”,”ratón”]
ListaN =[2, 1, 3]
print (“Método sort()”)
ListaN.sort() #ordenar ascendente numérica
ListaS.sort()
print (ListaN)
print (ListaS)
print()

#Método reverse() invierteel orden de la lista numerica o de string.
ListaS =[“perro”,”gato”,”ratón”]
ListaN =[2, 1, 3]
```

```
print (“Método reverse()”)
ListaN.reverse() #ordenar ascendente numérica
ListaS.reverse() #ordenar ascendente string
print (ListaN)
print (ListaS)
print()

#Método index() recibe como parámetro un objeto y devuelve el índice de su
primera aparición.
Lista =[“perro”,”gato”,”ratón”]
print (“Método index()”)
print(Lista.index(“ratón”)) #ordenar ascendente numérica
```

```
Método append()
[1, 2, 3, ‘perro’, ‘gato’, ‘ratón’, 40]

Método extend()
[1, 2, 3, ‘perro’, ‘gato’, ‘ratón’, ‘a’, ‘b’, ‘c’]
Método append()
[1, 2, 3, ‘perro’, ‘gato’, ‘ratón’, 40]

Método extend()
[1, 2, 3, ‘perro’, ‘gato’, ‘ratón’, ‘a’, ‘b’, ‘c’]

Método insert()
[1, ‘X’, 2, 3, ‘perro’, ‘gato’, ‘ratón’]

Método remove()
[1, 2, ‘perro’, ‘gato’, ‘ratón’]

Método sort()
[1, 2, 3]
[‘gato’, ‘perro’, ‘ratón’]

Método reverse()
[3, 1, 2]
[‘ratón’, ‘gato’, ‘perro’]

Método index()
2
```

3.3.5 Tuplas

En Python, las tuplas son una forma de estructura de datos que posibilita el almacenamiento de elementos de manera similar a las listas, pero con dos diferencias cruciales [49]. En primer lugar, las tuplas son inmutables, lo que implica que una vez que se declaran, no pueden ser modificadas. Segundo, a diferencia de las listas, las tuplas se inicializan utilizando paréntesis (). Estas particularidades hacen que las tuplas sean una elección apropiada en situaciones donde se necesita una estructura de datos que no cambie y que, en algunos casos, pueden ser más eficientes en términos de rendimiento en comparación con las listas. Por lo tanto, la elección entre usar listas o tuplas dependerá de las necesidades específicas de cada situación en programación.

```
# Crear una tupla
tupla = (1, 2, 3)
print(tupla)
print(tupla[0])
print( )

# Intentar modificar un  valor de una tupla. Se intenta modificar el valor
ubicado en la posición 0 por el valor de 8
tupla[0] = 8
print(tupla)

#Convertir una lista en tupla
Lista =[1, 2, 3, “perro”, “gato”, “ratón”]
tupla = tuple(Lista)
print (tupla)
```

```
(1, 2, 3)
1

# Error! TypeError

(1, 2, 3, “perro”, “gato”, “ratón)
```

3.3.6 Diccionarios

Los diccionarios en Python son una estructura de datos que permite el almacenamiento de información a través de pares llave-valor. En otras palabras, cada variable está vinculada a un valor y todas las llaves conforman un conjunto de datos. La creación de diccionarios se lleva a cabo mediante el uso de llaves {} y separando cada par key: valor con comas. Los diccionarios en Python poseen varias propiedades

distintivas como ser dinámicos, lo que significa que pueden crecer o reducir su tamaño, permitiendo la adición o eliminación de elementos según sea necesario; son indexados, lo que facilita el acceso a los elementos del diccionario utilizando su clave correspondiente, y son anidados, lo que implica que un diccionario puede incluir a otro diccionario en su campo de valor, lo que proporciona flexibilidad para estructurar datos de manera jerárquica y compleja.

```
#Conocer cuáles son las llaves y valores del diccionario
diccionario = {   'X1': 1,       'X2': 2,      'X3': 3   }
print("Las llaves del diccionario son: %s" % diccionario.keys())
print("Los valores del diccionario son: %s" % diccionario.values())
```

```
Las llaves del diccionario son:   dict_keys (['X1', 'X2', 'X3'])
Los valores del diccionario son: dict_values ([1, 2, 3])
```

Un aspecto interesante con los diccionarios es que en ellos es posible modificar las claves y los valores como se muestra en los siguientes fragmentos de código Pyhton. Para modificar un valor se deben usar [ ] con el nombre del key y asignar a esta llave el valor que se desea modificar.

```
#Conocer el diccionario base con llaves y valores
diccionario = { 'X1': 1, 'X2': 2, 'X3': 3 }
print(diccionario)

# Como acceder a un "valor" del diccionario
print("Estoy accediendo al valor de la llave X1:", diccionario['X1'])

# Agregar un valor nuevo a un diccionario. Por ejemplo agregar valor a la
llave X4
diccionario['X4'] = "perro"
print("El nuevo diccionario será", diccionario)

# Modificar un valor del diccionario. Por ejemplo modificar valor a la llave
X4
diccionario['X4'] = "sofia"
print("El nuevo diccionario será", diccionario)
```

```
{'X1': 1, 'X2': 2, 'X3': 3}
Estoy accediendo al valor de la llave X1: 1
El nuevo diccionario será {'X1': 1, 'X2': 2, 'X3': 3, 'X4': 'perro'}
El nuevo diccionario será {'X1': 1, 'X2': 2, 'X3': 3, 'X4': 'sofia'}
```

Algunos de los métodos usuales para trabajar con listas son los que se muestran en los siguientes fragmentos de código.

```
#Conocer el  diccionario base con llaves y valores
diccionario = { 'X1': 1, 'X2': 2, 'X3': 3 }
print(diccionario)

#Método get() consulta el valor para una llave (key) determinada
print(diccionario.get('X2'))
print()

#Método items() devuelve una lista con las llaves y valores del
diccionario.
items = diccionario.items()
print (items)
print()
#Método keys() devuelve una lista con todas las llaves (keys) del
diccionario
llaves = diccionario.keys()
print("Las llaves o keys del diccionario son:", llaves)
print()

#Método values () devuelve una lista con todos los valores (value) del
diccionario
valores= diccionario.values()
print("Las llaves o keys del diccionario son:", valores)
print()

#Método clear() elimina llaves y valores del diccionario
diccionario.clear()
print("Los datos que contiene el diccionario son:", diccionario)
print()
```

```
{'X1': 1, 'X2': 2, 'X3': 3}
2

dict_items([('X1', 1), ('X2', 2), ('X3', 3)])

Las llaves o keys del diccionario son: dict_keys(['X1', 'X2', 'X3'])

Las llaves o keys del diccionario son: dict_values([1, 2, 3])

Los datos que contiene el diccionario son: {}
```

3.3.7 Variables

El lenguaje Python es altamente sensible al uso de variables en minúsculas o mayúsculas. Esta sensibilidad se manifiesta claramente en los resultados obtenidos al ejecutar fragmentos de código de manera separada. En Python, las variables pueden crearse asignando un valor a un nombre sin necesidad de declararlas previamente. Es importante evitar el uso de palabras reservadas de Python (como True, list, etc.) y abstenerse de comenzar el nombre de la variable con espacios, guiones o números.

```
#Cuando las variables no están ambas en mayúsculas o en minúsculas
K= 1000
print(K)
print('------------')

#Cuando ambas variables están en mayúsculas o ambas en minúsculas
K= 1000
print('K vale…:', K)
print('-------------')

#Asignación múltiple de variables
x, y, z = 10, 20, 30
print (x)
print (y)
print (z)
```

```
NameError                 Traceback (most recent call last)
<ipython-input-40-9218024b9e4d> in <module>
      1 k= 1000
----> 2 print(K)
NameError: name 'K' is not defined
--------------------------------------
K vale…: 1000
--------------------------------------
10
20
30
```

3.3.8 Conversión de tipos de datos

En programación, realizar una operación de casting implica convertir un tipo de dato a otro. Por ejemplo, es factible tomar un valor que sea inicialmente una cadena de texto y transformarlo en un número entero o decimal. De manera análoga, se puede convertir un número entero en una cadena de texto para presentar información

de forma legible para el usuario. Este proceso de conversión es fundamental en programación, ya que posibilita adaptar los datos a las necesidades específicas de una aplicación. A continuación, se exploran algunas funciones que ofrece Python para llevar a cabo estas conversiones.

```
#Función float() convierte decimal a entero
k = 3.5
k = int(k)
print(k)
print ()

#Función string() convierte decimal a string
k = 3.5
print(type(k)) # la clase inicial es <class 'float'>
k = str(k)
print(type(k)) # la clase final es <class 'str'>
print(k)
print ()

#Función float() convierte string a decimal
k = "48.36"
print(float(k))
print ()

#Función string() convierte entero a string
k = 35
print(type(k)) # la clase inicial es <class 'float'>
k = str(k)

print(type(k)) # la clase final es <class 'str'>
print(k)
print ()
a = {1, 2, 3}
b = list(a)
print(type(a)) # <class 'set'>
print(type(b)) # <class 'list'>
```

```
3

<class 'float'>
<class 'str'>
3.5

48.36
```

```
<class 'int'>
<class 'str'>
35

<class 'set'>
<class 'list'>
```

3.3.9 Condicional IF

Cuando se ejecuta un código en cualquier lenguaje de programación, generalmente sigue un flujo secuencial, es decir, línea por línea en el orden establecido. Sin embargo, para proporcionar mayor flexibilidad y control, se emplean estructuras de control, como los condicionales (por ejemplo, if), bucles (for, while, etc.), entre otras [50].

Estas estructuras de control permiten modificar la ejecución del código, alterando su flujo secuencial, cambiando la dirección de ejecución del programa en función de condiciones específicas. Por ejemplo, ciertos bloques de código se ejecutan solo si se cumplen determinadas condiciones, lo que brinda la capacidad de adaptar el comportamiento del programa según las circunstancias.

If, es uno de los condicionales más sencillos utilizados en Python. Se escribe "if" y expresa "si". Hay que tener presente que se debe colocar " : " al final de la instrucción. Estos dos puntos quieren decir " entonces".

```
# La condición que se tiene que cumplir para que el bloque de código se
ejecute
# En este ejemplo, "if" comprueba si 20 es "igual" al resultado de 100/5

if  20 == 100/5:
    print('Respuesta correcta')
    print('-------------')
```

```
Respuesta correcta
-------------
```

Además del operador utilizado en la declaración condicional "if", existen otros operadores que posibilitan la comparación de dos números. El empleo de estos operadores sigue la misma lógica mencionada anteriormente, permitiendo evaluar diversas condiciones. En lugar de limitarse a comparar un número con una única condición, estos operadores posibilitan la comparación entre una o varias condiciones

simultáneamente. A continuación, se presenta un ejemplo que ilustra cómo estos operadores pueden utilizarse para realizar comparaciones más complejas.

```
# Ejemplo de uso de una instrucción condicional con división
a = 10
b = 2

# Verificación de la condición: si b es mayor que 0
if b > 0:
    resultado_division = a / b
    print("El resultado de la división es:", resultado_division)


# Ejemplo de uso de operadores en una declaración condicional
numero_1 = 10
numero_2 = 5
# Condición: si numero_1 es mayor que numero_2 y al mismo tiempo numero_2
es positivo
if numero_1 > numero_2 and numero_2 > 0:
    print("El primer número es mayor que el segundo y el segundo es
positivo.")
```

```
El resultado de la división es: 5.0
El primer número es mayor que el segundo y el segundo es positivo.
```

La instrucción "if" es comúnmente utilizada de manera concatenada en líneas de código para gestionar diversas condiciones de manera eficiente. En el siguiente ejemplo, se ilustra su aplicación en una combinación de condiciones, lo cual permite realizar acciones específicas.

```
# Ejemplo de uso de if en concadenados
x = 10
y = 'Hola'
z = 33
if z == 34 or y == 'Hola':
     x = x + 1 y = y + ' Estoy aprendiendo Python' # String concadenado.
     print (x)
     print (y)
```

```
13
Hola Estoy aprendiendo Python
```

3.3.10 Condicional ELSE

Cuando se utiliza la condición "if", el bloque de código asociado se ejecuta solo si se cumple esa condición. Sin embargo, ¿qué ocurre cuando se desea ejecutar un bloque de código cuando la condición no es verdadera? Aquí es donde entra en juego la opción "else". Esta declaración indica a la computadora que ejecute el siguiente conjunto de comandos si la condición anterior no es válida.

Para este ejemplo particular, el programa solicita al usuario que introduzca un número por teclado. Luego, se pregunta si la división de ese número da como resultado cero o si el número es impar. En caso de que no se cumpla ninguna de estas condiciones, se ejecutará el bloque de código asociado al "else".

```
# Ejemplo de uso de la instrucción if con else
numero = 7

# Condición: si el número es par
if numero % 2 == 0:
     print("El número es par.")
# Fin del bloque if # Si la condición del if no se cumple, se ejecuta el
bloque else
else:
      print("El número es impar.")
```

```
El número es impar.
```

Otro ejemplo de uso en cual se puede introducir el número al que desea determinar si es par o impar es el siguiente.

```
x=int(input("Introduzca un numero:"))
if x == 5:
      print("Es 5")
else:
      print("No es 5")
```

```
Introduzca un numero:4
No es 5
```

3.3.11 Condicional ELIF

Hasta el momento, hemos explorado la ejecución de bloques de código basándonos en si se cumple o no una instrucción específica. Sin embargo, en muchos

escenarios, es necesario manejar diversas condiciones, cada una con su propio conjunto de acciones asociadas. Es en este contexto que entra en juego el "elif" (abreviatura de "else if"), que permite incorporar condiciones adicionales al código. Podríamos traducir "elif" como "sino". Este condicional proporciona la capacidad de evaluar múltiples situaciones de forma secuencial y ejecutar el bloque de código asociado a la primera condición verdadera encontrada. A continuación, se presenta un ejemplo que demuestra el uso de "elif". En este ejemplo, la instrucción "elif" se utiliza para evaluar diferentes rangos de puntuación, proporcionando un enfoque escalonado para brindar mensajes específicos en función del rendimiento.

```
# Ejemplo de uso de la instrucción elif
puntuacion = 75

# Evaluación de múltiples condiciones utilizando elif
if puntuacion >= 90:
    print("Excelente desempeño. ¡Felicidades!")
# Fin del bloque if
elif 80 <= puntuacion < 90:
    print("Buen trabajo. Sigamos esforzándonos.")
# Fin del bloque elif
elif 70 <= puntuacion < 80:
    print("Aprobado, pero hay margen de mejora.")
# Fin del bloque elif
else:
    print("Lo siento, no has alcanzado la puntuación mínima requerida.")
# Fin del bloque else
```

```
Aprobado, pero hay margen de mejora.
```

Otro ejemplo de uso en cual se pueden introducir diferentes números (tres números dados x, y, z); y mediante un código que utiliza el condicional "if" se evalúa la condición principal, luego mediante "elif" se establecen dos condicionales (sino) y finalmente sino se cumplen alguna de los condicionales establecidos, se plantea la opción "else".

```
x=int(input("Introduzca un valor:"))
y=int(input("Introduzca un valor:"))
z=int(input("Introduzca un valor:"))

# Versión 1
if x>z:
    print("El mayor de los tres números es: %d" % x)
elif z>y:
```

```
            print("El mayor de los tres números es: %d" % z)
elif y>z:
    print("El mayor de los tres números es: %d" % y)
else:
    print("El mayor de los tres números es: %d" % z)

# Versión 2
print("El mayor de los tres números es: %d" % max(x,y,z))
```

```
Introduzca un valor:10
Introduzca un valor:15
Introduzca un valor:20

El mayor de los tres números es: 20
```

3.3.12 Ciclo con FOR

Existen tres tipos fundamentales de control de flujo que es crucial entender en programación: "if", "for" y "while". Hasta ahora, se ha explorado el condicional "if", que permite ejecutar bloques de código basándose en la evaluación de condiciones.

A continuación, se enfoca el texto en los bucles "for" y "while". Estos dos últimos son esenciales para la iteración, es decir, para repetir un bloque de código múltiples veces, pero difieren en su enfoque. El bucle "for" se utiliza para recorrer una secuencia de elementos, como una lista o una cadena de texto, mientras que el bucle "while" se repite mientras una condición específica sea verdadera. La comprensión de estos tres elementos proporciona una base sólida para estructurar el flujo de ejecución de un programa de manera efectiva. Se proporcionan ejemplos básicos de un bucle "for" para ilustrar su funcionamiento.

En este primer ejemplo, el bucle "for" recorre cada elemento en la lista de frutas e imprime un mensaje para cada una.

```
# Ejemplo de bucle for para recorrer una lista
frutas = ["manzana", "plátano", "uva"]
for fruta in frutas:
    print("Me gusta comer", fruta)
```

```
Me gusta comer manzana
Me gusta comer plátano
Me gusta comer uva
```

En Python, la capacidad de iterar se extiende a una amplia gama de estructuras de datos, y un ejemplo común es la iteración a través de una cadena de texto mediante el uso del bucle "for". En el siguiente ejemplo, se observa cómo la variable "i" adquiere sucesivamente los valores de cada letra en la cadena. Este proceso de iteración facilita la manipulación y análisis de datos en diversos contextos.

```
# Ejemplo de iteración sobre una cadena de texto en Python
cadena = "Python"
# Iterar sobre la cadena utilizando un bucle for
for letra in cadena:
    print("Letra actual:", letra)
```

```
Letra actual: P
Letra actual: y
Letra actual: t
Letra actual: h
Letra actual: o
```

Letra actual: n La capacidad de anidar bucles "for" en Python, es decir, colocar uno dentro de otro, resulta extremadamente útil cuando se requiere iterar sobre objetos que, a su vez, contienen otras estructuras iterables. Este escenario se presenta con frecuencia al manejar estructuras de datos compuestas, como listas de listas, que pueden interpretarse como matrices. El siguiente ejemplo ilustra la anidación de bucles "for" al recorrer una matriz representada por una lista de listas.

En este caso, el primer bucle "for" itera sobre cada fila de la matriz, y el segundo bucle "for" se encuentra dentro de este, iterando sobre cada elemento dentro de la fila correspondiente. Esta técnica de anidación es valiosa para abordar estructuras de datos más complejas y resalta la flexibilidad de Python al tratar con iterables anidados.

```
# Ejemplo de anidación de bucles for para iterar sobre una matriz (lista de
listas)
matriz = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]

# Anidación de bucles for para recorrer la matriz
for fila in matriz:
    for elemento in fila:
        print(elemento, end=" ")
    print() # Salto de línea para separar las filas en la salida
```

```
1 2 3
4 5 6
7 8 9
```

En el ejemplo siguiente se muestra la interacción de un ciclo “for” dentro de otro ciclo “for”. Para este ejemplo se imprime todas las tablas de multiplicar del 1 al 2. Cada iteración del primer bucle “for” representa el número base de la tabla de multiplicar, y el segundo bucle “for”, anidado en el primero, itera a través de los multiplicadores para generar los productos correspondientes.

```
# Ejemplo de ciclos for anidados para imprimir tablas de multiplicar del 1
al 2
for base in range(1, 3):  # Bucle externo para el número base
    print(f”Tabla de multiplicar del {base}:”)

    for multiplicador in range(1, 11):  # Bucle interno para los
multiplicadores
        resultado = base * multiplicador
        print(f”{base} x {multiplicador} = {resultado}”)

    print()  # Salto de línea para separar las tablas
```

```
Tabla de multiplicar del 1:
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10

Tabla de multiplicar del 2:
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

3.3.13 Función RANGE ( )

La función range() en Python crea una secuencia de números que, por defecto, comienza en 0 y se extiende hasta el número especificado como parámetro, excluyendo este último. Sin embargo, se pueden proporcionar hasta tres parámetros separados por comas: el primero indica el inicio de la secuencia, el segundo establece el final, y el tercero especifica el paso deseado entre los números. En situaciones estándar, donde no se especifican estos parámetros adicionales, se asume un inicio de 0 y un paso de 1. Es decir, la función range() generará una secuencia que inicia en 0 y avanza de uno en uno hasta el número final especificado. En este primer ejemplo, se muestra el ciclo for para imprimir todos los números desde 0 a al 9.

```
# Ejemplo de bucle for utilizando la función range()
#Este fragmento de código mostrará en la salida los números del 0 al 4
for i in range(5):
    print(i)
print('-------------- ')

#Este fragmento de código mostrará en la salida los números del 14 al 16
con saltos de 24
for i in range(14, 20, 2):
    print(i) #5,7,9,11,13,15,17,19
```

```
0
1
2
3
4
--------------
14
16
18
```

En el siguiente ejemplo, se emplea un ciclo "for" en conjunto con la función range(), acompañado de una variable llamada "suma" que inicializa en cero. Esta variable cumple la función de acumular la suma de los primeros 100 números generados por la secuencia proporcionada por range(). A medida que el bucle itera sobre estos números, la variable "suma" se actualiza en cada iteración, permitiendo así calcular la suma total al final del proceso.

```
#Ejemplo de uso de ciclo for y la función range() para sumar los primeros
100 números
suma = 0
```

```
for numero in range(1, 101):
    suma += numero
print("La suma de los primeros 100 números es:", suma)
```

```
La suma de los primeros 100 números es: 5050
```

3.3.14 Ciclo con WHILE

La instrucción "while"(usado como, "mientras que") en Python posibilita la ejecución repetida de un bloque de código, de ahí su nombre. Este bucle se mantiene activo mientras una condición específica sea verdadera. En el momento en que esta condición deje de cumplirse, el programa sale del bucle y procede con la ejecución normal del código. Cada ejecución completa del bloque de código dentro del "while" se denomina iteración, destacando la capacidad de este bucle para realizar operaciones iterativas de manera controlada. While es usado como "mientras que".

A continuación, se muestra un ejemplo de uso para imprimir los números pares hasta el 10. La condición contador <= 10 se evalúa al comienzo de cada iteración, y el contador se incrementa en cada ejecución del bloque (en +2). Este control preciso permite adaptar el comportamiento del bucle a situaciones donde la cantidad de iteraciones no es conocida de antemano.

```
# Ejemplo de uso del bucle while para contar mientras que no sea 10
contador = 0
while contador <= 10:
    print("Iteración:", contador)
    contador += 2
print("Fin del bucle while")
```

```
Iteración: 0
Iteración: 2
Iteración: 4
Iteración: 6
Iteración: 8
Iteración: 10
Fin del bucle while
```

3.3.15 Iterar con Zip

Iterar con la función zip() en Python proporciona una forma eficiente de recorrer simultáneamente múltiples iterables (por ejemplo: listas). La función zip()

combina elementos correspondientes de varias secuencias en tuplas y crea un iterador que produce estas tuplas en cada iteración.

Al utilizar un bucle "for" en conjunto con zip(), es posible acceder a los elementos emparejados de manera sincronizada, lo cual es particularmente útil cuando se trabaja con listas, tuplas u otros objetos iterables de longitud igual. Este enfoque facilita la manipulación de datos emparejados y es una herramienta valiosa en la programación Python.

En el ejemplo siguiente, zip(nombres, edades) combina los elementos correspondientes de las listas "nombres" y "edades", y el bucle "for" itera sobre estas tuplas emparejadas, permitiendo imprimir de manera eficiente la información relacionada.

```
# Ejemplo de iteración con zip en Python usando dos listas diferentes
nombres = ["Alice", "Bob", "Charlie"]
edades = [25, 30, 22]

# Utilizando zip() para emparejar elementos de dos listas
for nombre, edad in zip(nombres, edades):
    print(nombre, "tiene", edad, "años.")
    print (nombre, edad)
    print()
```

```
Alice tiene 25 años.
Alice 25

Bob tiene 30 años.
Bob 30

Charlie tiene 22 años.
Charlie 22
```

En el ejemplo anterior, se ilustró el uso de la función zip con dos listas; sin embargo, es importante destacar que esta función también puede utilizarse con varias listas, incluso cuando estas no tienen la misma cantidad de elementos. En este caso, las listas "nombres" y "edades" tienen la misma longitud, mientras que la lista "paises" tiene un elemento menos. A pesar de esta diferencia, la función zip maneja la iteración de manera armoniosa, emparejando elementos hasta la longitud más corta y permitiendo el acceso a la información correspondiente. Este comportamiento versátil de zip ofrece flexibilidad al trabajar con conjuntos de datos de longitud variable.

```
# Ejemplo de uso de zip con múltiples listas de longitud variable
nombres = [“Pedro”, “Maria”, “Beto”]
edades = [25, 30, 22]
paises = [“EE. UU.”, “Colombia”]

# Utilizando zip() para emparejar elementos de tres listas
for nombre, edad, pais in zip(nombres, edades, paises):
    print(nombre, “tiene”,edad, “años y es de”, pais)
```

```
Pedro tiene 25 años y es de EE. UU.
Maria tiene 30 años y es de Colombia
```

Hasta este punto, hemos explorado el uso de la función zip exclusivamente con listas; sin embargo, es fundamental destacar que esta función no está limitada a listas y puede aplicarse a cualquier clase iterable. Por ende, podemos emplear zip con diccionarios. En este ejemplo zip se adapta sin problemas, emparejando las claves o llaves de los diccionarios en cada iteración.

```
# Ejemplo de uso de zip con diccionarios SOLO para la llave
Dicc_1= {‘Colombia’: ‘Uno’,
         ‘España’: ‘Dos’,
         ‘Nigeria’: ‘Tres’}
Dicc_2 = {‘-Venezuela’: ‘One’,
          ‘-Portugal’: ‘Two’,
          ‘-Senegal’: ‘Three’}

# Utilizando zip() para emparejar las llaves de los dos diccionarios
for a, b in zip(Dicc_1, Dicc_2):
    print(a, b)
```

```
Colombia -Venezuela
España -Portugal
Nigeria -Senegal
```

Surge la pregunta de cómo se podría capturar y manipular tanto las claves como los valores de cada elemento dentro de los diccionarios. En respuesta a esta interrogante, entra en juego la función items(), que nos brinda la capacidad de acceder a la clave y al valor de cada elemento de un diccionario. Esta función se convierte en una herramienta esencial al trabajar con diccionarios, permitiendo un acceso sencillo y directo a la información almacenada.

```
# Ejemplo de uso de zip y de la función items() para acceder a llaves o
claves y valores de un diccionario
Dicc_1= {‘Colombia’: ‘Uno’, ‘España’: ‘Dos’, ‘Nigeria’: ‘Tres’}
Dicc_2 = {‘-Venezuela’: ‘One’, ‘-Portugal’: ‘Two’, ‘-Senegal’: ‘Three’}
for (k1, v1), (k2, v2) in zip (Dicc_1.items(), Dicc_2.items()):
    print(k1, v1, v2)

print()
for (k1, v1), (k2, v2) in zip(Dicc_1.items(), Dicc_2.items()):
    print(“pais”,k1, “vecino de”, k2, “es el numero”, v1)
```

```
Colombia Uno One
España Dos Two
Nigeria Tres Three

pais Colombia vecino de -Venezuela es el número Uno
pais España vecino de -Portugal es el numero Dos
pais Nigeria vecino de -Senegal es el número Tres
```

3.3.16 Iterar con Enumerate

La función enumerate() en Python proporciona una forma eficaz de iterar sobre una secuencia mientras se realiza un seguimiento del índice o posición de cada elemento. Al utilizar enumerate(), obtenemos tuplas que contienen tanto el índice como el valor correspondiente durante cada iteración. En este ejemplo, enumerate() permite acceder a la posición (índice) y al valor de cada elemento en la lista frutas. Esta función es valiosa en situaciones donde es necesario realizar un seguimiento explícito del índice durante la iteración.

```
# Ejemplo de uso de la función enumerate()
frutas = [“manzana”, “plátano”, “uva”]

# Utilizando enumerate() para obtener índices y valores
for indice, fruta in enumerate(frutas):
    print(f”Índice {indice}: {fruta}”)
```

```
Índice 0: manzana
Índice 1: plátano
Índice 2: uva
```

3.3.17 Funciones

Python como otros lenguajes usan diferentes funciones dentro de un programa. Para describir una función se usa la palabra clave "def" seguidamente se escribe el nombre de la función que se va a usar. Acto seguido añadimos paréntesis. Técnicamente dentro de los paréntesis podemos colocar los parámetros de la función en caso de que sea requerido [46], los cuales son los argumentos de entrada. Asi mismo una función posee un código a ejecutar y unos parámetros de salida.

Para nuestro ejemplo vamos a utilizar el nombre de "función_simple" ya que crearemos una función muy básica. Así mismo para una función es necesario dejar un espacio (sangrado) en la siguiente línea de código. El siguiente ejemplo define una nueva función para calcular la suma de dos valores y llama a la función con dos argumentos.

A continuación, se procede a describir una función simple en Python, es decir sin argumento y se muestra cómo se llama a la función. Llamar a la función quiere decir cómo se le dice al programa que vaya hasta donde la función, ejecute las instrucciones que en ella se encuentran y regrese a ejecutar el código por fuera de la función.

```
def funcion_simple():
    print("Esta es una función simple sin argumentos")

#De esta forma se llama a la función
funcion_simple()
```

Una forma de comprender cómo trabaja una función es tener en cuenta cómo lo hace una función en matemáticas (y =F(x)), donde la variable *y* tendrá un valor dependiendo el valor de *x* (argumento) que se le entregue a la función. Y es sabido, que cuando se ingresa el valor de *x* a la función esta realiza un proceso matemático para obtener la respuesta de *y*.

A modo de ejemplificación, se presentan dos casos de uso de funciones en Python. En el primer ejemplo, se define una función de fusión que, al recibir un argumento representado por la variable X, realiza una operación multiplicativa por 5, generando así el resultado almacenado en la variable Y. Por otro lado, el segundo ejemplo presenta una función que, al ser proporcionada con los argumentos (a, b), produce un valor Y que corresponde a la suma de ambos valores.

Los argumentos posicionales, representan la forma más fundamental e intuitiva de pasar parámetros a una función en Python. Supongamos que tenemos una función llamada fusión () que espera un parámetro y una fusión llamada suma

que espera dos parámetros, entonces dichas funciones pueden ser invocada de la siguiente manera.

```
#Primer ejemplo: Función de fusión
def fusion(x):
    y = x * 5
    return y

# Segundo ejemplo: Función de suma
def suma(a, b):
    y = a + b
    return y

# Llamada a la función fusion con un argumento
resultado_fusion = fusion(10)
print("Resultado de la multiplicación fue:", resultado_fusion)

# Llamada a la función suma con dos argumentos
resultado_suma = suma(10, 20)
print("Resultado de la suma fue:", resultado_suma)
```

```
Resultado de la multiplicación fue: 50
Resultado de la suma fue: 30
```

La utilización de la sentencia "return" en una función en Python cumple con dos propósitos esenciales: en primer lugar, facilita la salida de la función, permitiendo la transferencia de la ejecución de vuelta al punto donde se realizó la llamada. En segundo lugar, la sentencia "return" posibilita la devolución de uno o varios valores como resultado de la ejecución de la función. Esto es esencial cuando se desea utilizar o procesar la salida de la función en el contexto más amplio del programa.

Las funciones en Python pueden incorporar diversas estructuras condicionales, y la sentencia "return" se convierte en un elemento clave para ofrecer diferentes salidas en función de las condiciones evaluadas. En este contexto, se puede diseñar una función que, al recibir un número como parámetro, determine si dicho valor es mayor, menor o igual a 20. La sentencia "return" se empleará para comunicar el resultado de la evaluación y permitir que el programa principal acceda a esta información de manera clara y estructurada.

```
# Ejemplo de función con condicionales y sentencia return
def comparar_con_20(numero):
    if numero > 20:
```

```
            return numero, “es mayor que 20”
    elif numero < 20:
            return numero, “es menor que 20”
    else:
            return numero, “es igual a 20”

# Llamada a la función con diferentes valores
resultado1 = comparar_con_20(25)
resultado2 = comparar_con_20(15)
resultado3 = comparar_con_20(20)
print(resultado1)
print(resultado2)
print(resultado3)
```

```
(25, ‘es mayor que 20’)
(15, ‘es menor que 20’)
(20, ‘es igual a 20’)
```

Además, en Python, la sentencia “return” brinda la capacidad de devolver más de una variable, las cuales se separan por comas. En el siguiente ejemplo, se presenta una función que calcula la suma y la media de tres números, proporcionando ambos resultados como salida. En este caso, la función calcular_suma_y_media() realiza los cálculos necesarios y utiliza la sentencia “return” para devolver tanto la suma como la media de los tres números. Al recibir la llamada a la función, se pueden asignar ambas variables resultantes, permitiendo un acceso conveniente a ambas salidas de manera individual. Esta funcionalidad añade versatilidad a la hora de diseñar funciones que necesitan proporcionar múltiples resultados para su posterior utilización.

```
# Ejemplo de función que devuelve múltiples variables con la sentencia
return
def calcular_suma_y_media(num1, num2, num3):
    suma = num1 + num2 + num3 media = suma / 3
    return suma, media

# Llamada a la función con diferentes valores
resultado_suma, resultado_media = calcular_suma_y_media(10, 15, 20)
print(f”La suma es: {resultado_suma}”)
print(f”La media es: {resultado_media}”)
```

```
La suma es: 45
La media es: 15.0
```